Comparing and Combining Genetic and Clustering Algorithms for Software Component Identification from Object-Oriented Code

Selim Kebir
LIRMM
University of Monptellier 2
France
smkebir@gmail.com

Abdelhak-Djamel Seriai
LIRMM
University of Monptellier 2
France
seriai@lirmm.fr

Sylvain Chardigny
MGPS
Port-Saint-Louis
France
s.chardigny@mgps.info

Allaoua Chaoui
MISC Laboratory
University of Constantine
Algeria
a_chaoui2001@yahoo.com

ABSTRACT

Software component identification is one of the primary challenges in component based software engineering. Typically, the identification is done by analyzing existing software artifacts. When considering object-oriented systems, many approaches have been proposed to deal with this issue by identifying a component as a strongly related set of classes. We propose in this paper a comparison between the formulations and the results of two algorithms for the identification of software components: clustering and genetic. Our goal is to show that each of them has advantages and disadvantages. Thus, the solution we adopted is to combine them to enhance the results.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Components, Algorithms

Keywords

Software component identification, hierarchical clustering, genetic algorithm

1. INTRODUCTION

Software component identification is one of the primary research problems in CBSE [4]. When applied to object oriented systems, software component identification result is a set of components where each one contains a set of classes. On the one hand, the relationship between classes belonging to a component must be high. On the other hand, the relationship between classes belonging to two different components must be low. Otherwise, software component identification can be performed in two different manners [4]: top-down and bottom-up. Top-down software component identification is performed by analyzing domain business models to get a set of business components. Bottom-up software component identification is performed by extracting reusable software components from existing software system source code.

In our previous works [5][6], we have defined a semantic-correctness and quality model for extracting software architectures from object-oriented source code. In this paper, we rely on our previous results to propose two algorithms for software component identification. The former is based on hierarchical clustering. The latter is based on genetic algorithm. Based on the results obtained by applying them on large-scale systems, we discuss the advantages and shortcomings of each one. Then, we propose a more efficient solution to identify software components by combining the results obtained by the two previous algorithms.

This paper is organized as follows: Section 2 introduces the mapping model between object and component concepts and the measurement model used to evaluate the semantic-correctness of a component. In section 3, we formulate the component identification problem as a partitioning problem. Based on the above formulation, we describe respectively in section 4 and section 5 how components can be identified using hierarchical clustering and genetic algorithms. In section 6, we present a comparison between hierarchical clustering and genetic algorithm in the context of software component identification. Next, we explain how we can combine these two algorithms to enhance the quality of the results followed by the experimentation of our approach on different systems of various sizes, first by applying separately hierarchical and genetic algorithms and next by combining them. In section 7, we present related works and we compare them with the our. Section 8 concludes the paper.

C^3*S*^{2}*E-12 2012* June 27-29, Montreal [QC, CANADA]
Editors: B. C. Desai, S. Mudur, E. Vassev

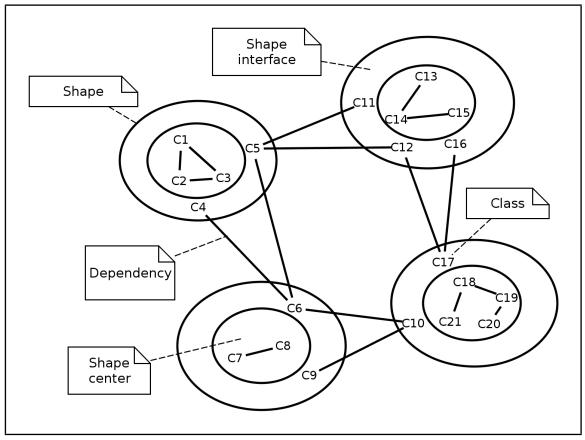

Figure 1: Component Structure

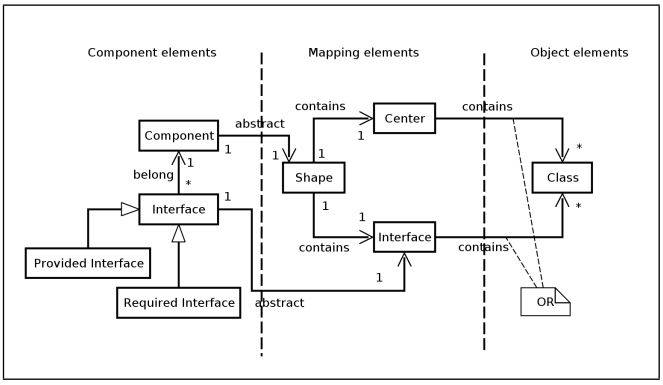

Figure 2: Object-Component Mapping Model

2. FROM OBJECT METRICS TO COMPONENT QUALITY CHARACTERISTICS

To identify software components from object oriented code, we rely on an adaptation of two elements presented in our previous works [5][6]: Firstly, A mapping model between object oriented concepts (i.e. classes, interfaces, packages, etc) and component based software engineering ones (i.e. components, interfaces, sub-component, etc.). Secondly, A measurement model of quality and semantic correctness of a component. This model refines characteristics of a component to measurable metrics. Based on these metrics, we define a fitness function to measure the quality and the semantic-correctness of a component.

2.1 Mapping model between component and object concepts

We define components as disjoint collections of classes. These collections are named "shapes" and contain classes which can belong to different object oriented packages (Figure 1). Each shape is composed of two sets of classes: the "shape interface" which is the set of classes which have a link with some classes from the outside of the shape, e.g. a method call or attribute use from the outside; and the "center" which is the remainder of shape. As shown in figure 1, we assimilate component interface set to "shape interface" and component to shape. Figure 2 shows the component-object mapping model that we propose to handle the correspondence between object and component concepts.

2.2 Semantic-correctness of components

In order to measure the component semantic-correctness, we study component characteristics. This study is based on the most commonly admitted definitions of software component. Many definitions exist where each one characterizes a component somewhat differently. Nonetheless, some important commonalities exist among the most prevalent definitions.

Szyperski defines, in [17], a component as a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. In [9], Heinemann and Councill define a component as a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. Finally, Luer, in [15], makes a distinction between component and deployable component. He defines a component as a software element that (a) encapsulates a reusable implementation of functionnality, (b) can be composed without modification, and (c) adheres to a component model. A deployable component is a component that is (a) prepackaged, (b) independently distributed, (c) easily installed and uninstalled, and (d) self- descriptive.

By combining and refining the common elements of these definitions and others commonly accepted, we propose the following definition of a component:

A component is a software element that (a) can be composed without modification, (b) can be distributed in an autonomous way, (c) encapsulates an implementation of functionality, and (d) adheres to a component model.

Our component definition references component model. In our approach, the definition of a component model is the Luer one [15]: a component model is the combination of (a) a component standard that governs how to construct individual components and (b) a composition standard that governs how to organize a set of components into an application and how those components globally communicate and interact with each other.

As compared to the definitions of Luer and Heineman and Councill, we intentionally do not include the criterion that a component must adhere on a composition theory and the self-descriptive, pre-packaged and easy to install and uninstall properties of component. These are covered through the criterion that a component must adhere to a component model and does not need to be repeated.

In conclusion, according to our software component definition, we identify three semantic characteristics of software components: composability, autonomy and specificity.

2.2.1 From Characteristics to Properties

In the previous section, we have identified three semantic characteristics that we propose to evaluate. To do so, we adapt the characteristic refinement model given by the norm ISO-9126 [11]. According to this model, we can measure the characteristic semantic-correctness by refining it in the previous three semantic characteristics which are consequently considered as subcharacteristics. Based on the study of the semantic sub-characteristics, we refine them to a set of component measurable properties. Thus,

- A component is autonomous if it has no required inter-

face. Consequently, the property number of required interfaces should give us a good measure of the component autonomy.

- Then, a component can be composed by means of its provided and required interfaces. However, component will be more easily composed with another if services, in each interface, are cohesive. Thus, the property average of service cohesion by component interface should be a correct measure of the component composability.

- Finally, the evaluation of the number of functionalities is based on the following statements. Firstly a component which provides many interfaces may provide various functionalities. Indeed each interface can offer different services. Thus the higher the number of interfaces is, the higher the number of functionalities can be. Secondly if interfaces (resp. services in each interface) are cohesive (i.e. share resources), they probably offer closely related functionalities. Thirdly if the code of the component is closely coupled (resp. cohesive), the different parts of the component code use each other (resp. common resources). Consequently, they probably work together in order to offer a small number of functionalities. From these statements, we refine the specificity sub characteristic to the following properties: number of provided interfaces, average of service cohesion by component interface, component interface cohesion and component cohesion and coupling.

2.2.2 From Properties to Metrics

We cannot define the metrics which measure these properties on shapes. Consequently, according to our quality measurement model, we link the component properties to shape measurable properties.

- Firstly, according to our mapping model, component interface set is linked to the shape interface. As a result the average of the interface-class cohesion gives a correct measure of the average of service cohesion by component interface.

- Secondly the component interface cohesion, the internal component cohesion and the internal component coupling can respectively be measured by the properties interface class cohesion, shape class cohesion and shape class coupling.

- Thirdly in order to link the number of provided interfaces property to a shape property, we associate a component provided interface to each shape-interface class having public methods. Thanks to this choice, we can measure the number of provided interfaces using the number of shape interface classes having public methods.

- Finally, the number of required interfaces can be evaluated by using coupling between the component and the outside. This coupling is linked to shape external coupling. Consequently, we can measure this property using the property shape external coupling. In order to measure these properties, we need to define metrics.

The properties *shape class coupling* and *shape external coupling* require a coupling measurement. We define the metric $Coupl(E)$ which measures the coupling of a shape E and $CouplExt(E)$ which measures the coupling of E with the rest of classes. They measure three types of dependencies between objects: method calls, use of attributes and parameters of another class. Moreover they are percentages and are related through the equation: $couplExt(E) = 100 coupl(E)$. Due to space limitations, we do not detail these metrics. Shape properties *average of interface-class cohesion*, *interface-class cohesion*, and *shape-class cohesion* require a cohesion measurement. The metric Loose Class Cohesion (LCC), proposed by Bieman and Kang [3], measures the percentage of pair of methods which are directly or indirectly connected. Two methods are connected if they use directly or indirectly a common attribute. Two methods are indirectly connected if a connected method chain connects them. This metric satisfies all our needs for the cohesion measurement: it reflects all sharing relations, i.e. sharing attributes in object oriented system, and it is a percentage. Consequently, we use this metric to compute the cohesion for these properties. The refinement model is summarized in figure 3.

2.2.3 Evaluation of semantic-correctness

According to the links previously established between the sub- characteristics and the shape properties, we define the evaluation functions Spe, A and C respectively for specificity, autonomy and composability, where $nbPub(I)$ is the number of interface classes having a public method and $|I|$ is the cardinality of the shape interface I:

- $Spe(E) = \frac{1}{5} \cdot (\frac{1}{|I|} \cdot \sum_{i \in I} LCC(i) + LCC(I) + LCC(E) + Coupl(E) + nbPub(I))$
- $A(E) = couplExt(E) = 100 - coupl(E)$
- $C(E) = \frac{1}{|I|} \cdot \sum_{i \in I} LCC(i)$

The evaluation of the semantic-correctness characteristic is based on the evaluation of each sub-characteristic. That is why we define a local fitness function to measure the semantic-correctness of one component as a linear combination of each fitness function of sub-characteristics (i.e. Spe, A, and C):

$$S(E) = \frac{1}{\sum_i \lambda_i} \cdot (\lambda_1 \cdot Spe(E) + \lambda_2 \cdot A(E) + \lambda_3 \cdot C(E))$$

This form is linear because each of its parts must be considered uniformly. The weight associated with each function allows the software architect to modify, as needed, the importance of each sub-characteristic.

3. SOFTWARE COMPONENT IDENTIFICATION PROBLEM

After defining what a component is and how to measure its semantic- correctness using a fitness function, we have to formulate the software component identification problem as a partitioning problem where the resulting partition must contain as many good-quality components as possible. To do this, we define in the following the properties and the constraints that a partition must meet followed by the definition of a global fitness function to measure the overall quality of a set of identified components.

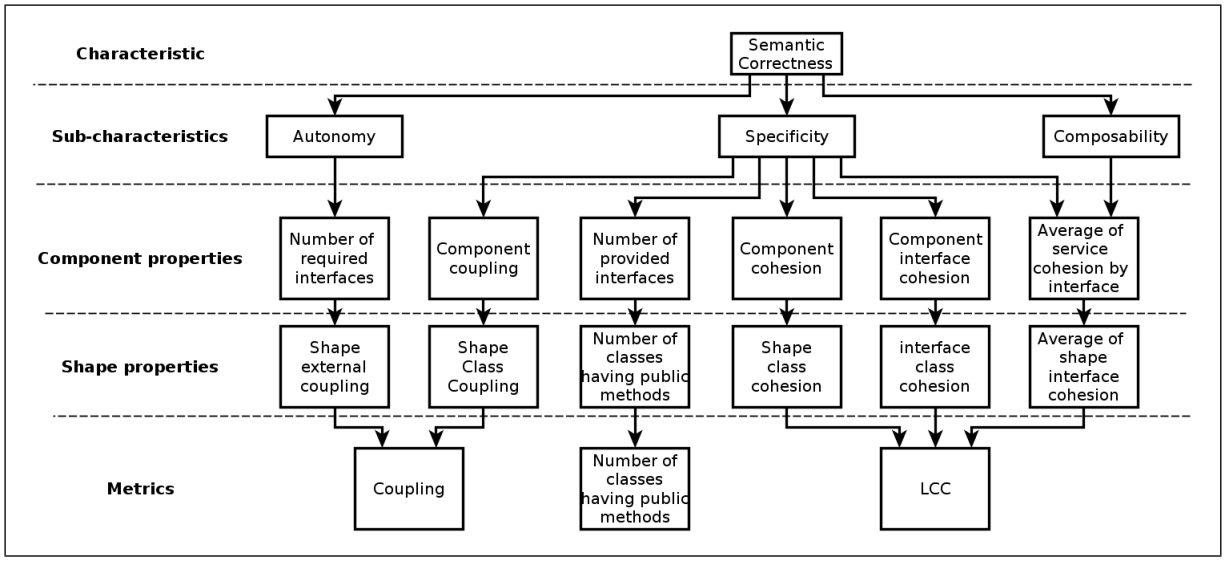

Figure 3: The refinement model for semantic-correctness

3.1 Software Component Identification as a partitionning problem

According to the mapping model between component and object concepts, the content of a component matches a set of classes. Thus, in order to define the sets of classes that can belong to a component, it is necessary to define a process for grouping these classes. This association must be based on a number of criteria to maximize the value of the above defined fitness function in these groups.

Hence, the component identification problem can be formulated as a partitioning problem where the input is a set of classes $C = \{c_1, c_2, ..., c_n\}$ and the output is a partition of C noted $P(C) = \{p_1, p_2, ..., p_k\}$ where:

- c_i is a class belonging to the system.
- p_i is a subset of C.
- k is the number of identified components.
- $P(C)$ does not contain empty elements: $\forall p_i \in P(C), p_i \neq \phi$
- The Union of all $P(C)$ elements is equal to C: $\bigcup_{i=1}^{k} p_i = C$. This property is called **completness**.
- The elements of $P(C)$ are pair wise disjoint:$\forall i \neq j, p_i \cap p_j = \phi$. This property is called **consistency**.

3.2 Definition of the global fitness function

The main purpose of software component identification techniques is to promote the reuse of recovered components as building blocks for newly developed systems. Thus, the identified components must exceed a certain threshold of quality to be potential candidates for reuse. Based on this, we define a global fitness function F to measure the quality of a partition P. This function returns the proportion of components for which the value of the local fitness function S exceeds a given threshold t chosen by the architect.

$$F(P,t) = \frac{\{c_i \in P, where\, S(c_i) > t\}}{|P|}$$

4. HIERARCHICAL CLUSTERING BASED TECHNIQUE FOR COMPONENT IDENTIFICATION

Clustering approaches can be classified as hierarchical or non-hierarchical. Hierarchical clustering techniques are further divided into agglomerative and divisive techniques. An agglomerative method involves a series of successive mergers whereas a divisive method involves a series of successive divisions [13].

4.1 Building a Hierarchy of Clusters

The approach proposed here makes use of a hierarchical agglomerative clustering algorithm for grouping classes. The strength of the relationship between the classes is used as basis for clustering them. This strength is measured using the fitness function defined previously. We propose the following algorithm to build a dendrogram from a set of classes.

Algorithm 1 HierarchicalClustering(file code):Tree dendro

```
classes ← extractInformation(code);
clusters ← classes;
while (|clusters| > 1) do
  (c1, c2) ← nearestClusters(clusters);
  c3 ← Cluster(c1, c2);
  remove(c1, clusters);
  remove(c2, clusters);
  add(c3, clusters);
end while
dendro ← get(0, clusters);
return dednro;
```

The technique proceeds through a series of successive binary mergers (agglomerations), initially of individual entities (classes) and later of clusters formed during the previous stages. The classes having the highest relationship strengths are grouped first. The process continues until a

cut-off point is reached. We obtain from this single cluster a dendrogram which represents the shape hierarchy. This dendrogram contains all candidate components. The presented algorithm uses the $nearestClusters()$ function to determine which two clusters will be merged in the next step. This function returns the most similar pair of clusters (the two clusters that maximize the value of the fitness function).

4.2 Selection of potentially good components

In order to obtain a partition of classes, we have to select nodes among the hierarchy resulting from the previous step. This selection is done by an algorithm based on a depth-first search which selects nodes representing shapes which will be the best components (Algorithm 2).

Algorithm 2 DendrogramTraversal(file code):Tree dendro

```
stack traversalClusters;
push(root(dendro), traversalClusters);
while (!empty(traversalClusters)) do
  Cluster father = pop(traversalClusters);
  Cluster f1 = son1(father, dendro);
  Cluster f2 = son2(father, dendro);
  if (F(father) > average(F(f1), F(f2))) then
    add(father, R);
  else
    push(f1, traversalClusters);
    push(f2, traversalClusters);
  end if
end while
return R;
```

For each node, we compare the result of the fitness function for the node shape and its sons. If the node result is inferior to the average of the result of its two sons, then the algorithm continues on the next node, else the node is identified to a shape and added to the partition and the algorithm computes the next node. In this way, good quality components will be identified while the traversal continues.

5. GENETIC BASED TECHNIQUE FOR COMPONENT IDENTIFICATION

Genetic algorithms are meta-heuristics inspired from the evolution theory of Darwin [10]. This meta-heuristic is used to obtain nearly optimal solutions to optimization problems by mimicking biological mechanisms such as, selection, crossover and mutation. Thus, to define a genetic algorithm for a given optimization problem, we must define four elements:

- A genetic representation of what a solution is.
- A fitness function to measure the quality of a solution.
- How to perform genetic operators.
- Content of the initial population.

We propose a genetic based algorithm for software component identification. We choose to randomly generate the content of initial population. We use the above-defined global fitness function of components as a fitness function for the genetic algorithm. Thus we only need to define the two remaining elements.

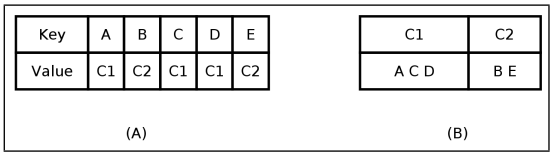

Figure 4: Representation of chromosomes

5.1 Genetic representation of components

Genetic algorithms start from an initial population which consists of a set of randomly generated solutions. Each solution has a genetic representation called chromosome. A chromosome is composed of genes that uniquely identify it. Based on this representation, genetic algorithms perform a series of successive iterations to evolve towards a better solution. The quality of a solution is evaluated using a fitness function. In our approach it is evaluated using the above-defined global fitness function. Thus, we need to define an adequate genetic representation of what a solution is.

In the software component identification problem, a solution is a given partition of all the system classes (cf. Section 3) where each element represents an identified component. Thus, a genetic representation of a solution must adhere to this definition and should allow us to know which class belongs to which component.

This can be performed in two manners; the first manner (Figure 4.a) consists in defining a chromosome as an associative array composed of a collection of genes noted (key,value) where each key is a class and each value is the component to which it belongs. Such that, each key appears only once in the collection. The second manner (Figure 4.b) consists in defining a chromosome as a partition of all classes. This partition is a set of non-overlapping and non- empty genes consisting in subsets that cover all classes.

We opt for the second representation because it allows verifying completeness and consistency properties more efficiently (cf. Section 3).

5.2 Genetic Operators

In order to converge towards best solutions, a genetic algorithm performs on the population a series of genetic operators. These operators allow creating new solutions by combining and modifying the current one. Next, we define selection, crossover and mutation operators based on the genetic representation that we have chosen above.

5.2.1 Selection

The first operator performed by a genetic algorithm on a population is the selection of individuals that will be combined to generate the next solutions. This selection can be performed in different manners (e.g. tournament, rank, steady-state, etc.). We opt for the roulette-wheel selection to favor the selection of the best solutions and to give opportunities to the other solutions to be selected for crossover.

5.2.2 Crossover

Crossover operators are used by genetic algorithms to explore the neighboring solutions by combining the selected ones.

Based on the genetic representation of components that we have defined previously, we can define a simple crossover

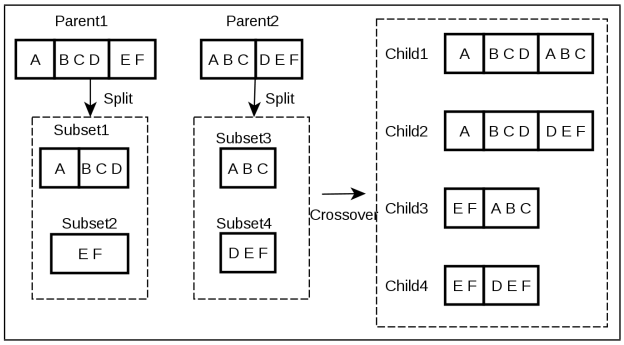

Figure 5: Simple crossover

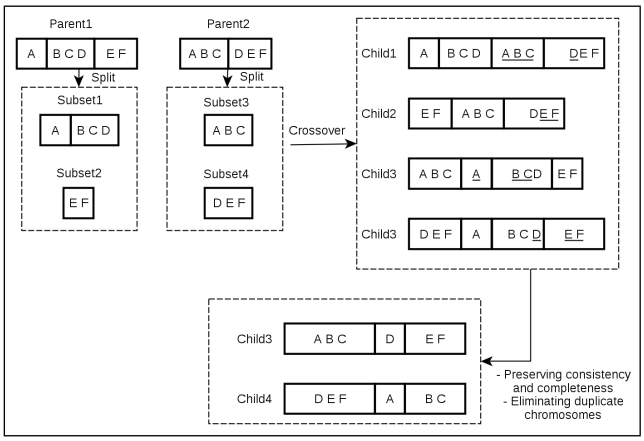

Figure 6: Preserving consistency and completeness

operator that combine two parents chromosomes to obtain new ones. The operator splits the chromosome of each parent into two subsets of genes and then combines each subset to obtain four children. This is illustrated in figure 5. As illustrated in figure 5, by using the simple crossover operator, we obtain children that violate the consistency and completeness properties. To avoid this problem, we define a crossover operator based on the one proposed in [8] to preserve these two properties. As illustrated in figure 6, to preserve completeness, we create a child by adding a subset of genes of the first parent to the second parent. Then to preserve consistency, we eliminate classes (Underlined in figure 6) that belong to new genes from the old genes and we remove eventually empty genes and duplicate chromosomes.

5.2.3 Mutation

Genetic algorithms use mutation operators to avoid limiting the exploration only in the neighborhood of the current population. This operator consists in modifying randomly a given chromosome. Based on the genetic representation of components that we have defined previously, we define a mutation operator that consists in performing randomly one of these two operations: fusion and separation. The former merges two randomly selected genes of a chromosome. The latter separates a randomly selected gene into two disjoint genes.

5.2.4 Selection of the next generation

After applying the above genetic operators, we obtain a new population containing the chromosomes of the previous iteration and newly generated chromosomes from crossovers and mutations. Therefore the size of the population increases at each generation of the algorithm. In order to keep the size of the population constant after applying crossovers and mutations, we use a second selection operator based on the roulette-wheel selection to obtain the next generation.

6. THE COMBINED APPROACH MOTIVATION

We show in this section the limits and the shortcomings of clustering and genetic approach taken separately. Then we propose to combine them.

6.1 Motivations of the combined approach

The main advantage of hierarchical clustering over genetic algorithm is its non-stochastic nature and low complexity when applied to medium systems. In fact, the complexity of the hierarchical clustering depends linearly on the number of classes of the system. However, the complexity of the genetic algorithm is high due to the complexity of the crossover and mutation operators. Also, the complexity of genetic algorithm depends highly on the size and the quality of the initial population.

Unlike genetic algorithm, hierarchical clustering provides a hierarchy of the identified components. Thus, it becomes possible to identify composite components. This offers the possibility to the architect to replace a component by its sub-components.

Even if hierarchical clustering does not perform an optimization of the global fitness function, in the case where relationships and dependencies between classes are relatively strong, it can identify good quality components by optimizing the local fitness function. In contrast, genetic algorithm tries to optimize the global fitness function using selection, crossover and mutation operators. However, on large-scale systems, a randomly generated initial population may produce bad results because the exploration performed by the genetic algorithm will spent lot of time before reaching good solutions. According to the previous comparison, using separately hierarchical clustering and genetic algorithms to identify software components have important limitations.

In order to overcome these limitations, we propose to combine the two algorithms to benefit from the advantages of each one. Instead of using a randomly generated initial population, we choose a population that contains the solution obtained by hierarchical clustering.

6.2 Experiment and results

In order to validate our hypothesis, we have applied hierarchical clustering (HC), genetic (GA) and combination of the two (HC-GA) component identification techniques on three different systems of different size (small, medium and large). The first is Apache Commons Email[1], a library that aims to provide an API for sending emails. It contains 44 classes. The second system is JDOM[2], a well-known library that provide a complete, Java-based solution for accessing, manipulating, and outputting XML data from Java code. It

[1]http://commons.apache.org/email/
[2]http://www.jdom.org/

contains 139 classes. The third system is Apache Http Components[3]; a toolset of low level Java components focused on HTTP and associated protocols. It contains 368 classes. We have chosen the value 0.7 for the threshold t of the global fitness function.

Tables 1, 2 and 3 summarize the obtained results for each system. Due to space limitations we use the following acronyms :

- **NIC**: Number of Identified Components.
- **GFF**: Global Fitness Function value.
- **NCET**: Number of Components Exceeding Threshold.

Table 1: Results on Common Email

	NIC	GFF	NCET
HC	5	0.20	1
GA	5	0.80	4
HC-GA	3	0.66	2

Table 2: Results on JDOM

	NIC	GFF	NCET
HC	11	0.36	4
GA	38	0.78	30
HC-GA	14	0.92	96

Table 3: Results on HTTP Components

	NIC	GFF	NCET
HC	34	0.55	19
GA	126	0.41	52
HC-GA	26	0.96	25

In order to compare the obtained results in terms of the optimization of the global fitness function, we place them on a bar diagram given in figure 7.

On a small system (commons email), genetic algorithm gives a better result than hierarchical clustering because of the following reasons: Firstly, when the number of classes is limited, the number of identified components decreases so the number of components which exceeds the quality threshold will be small. Secondly, the hierarchical clustering does not perform global optimization (cf. Section 6.1) unlike genetic algorithm which tries to globally enhance the quality of the solution at each generation. Finally, the small number of classes increases the probability to generate randomly a good initial population, so the exploration will likely lead to good-quality solutions after only few iterations.

Compared to genetic algorithms, the combined approach does not give a better result on a small system because it begins with a bad initial population so it have less chances to reach a better solution.

On a medium system (JDOM), genetic algorithm gives better results than hierarchical clustering for the same reasons cited above. However, even if hierarchical clustering does not perform global optimization, the third of components identified exceeds the quality threshold. This is explained by the significant number of classes contained in the system.

[3]http://hc.apache.org/

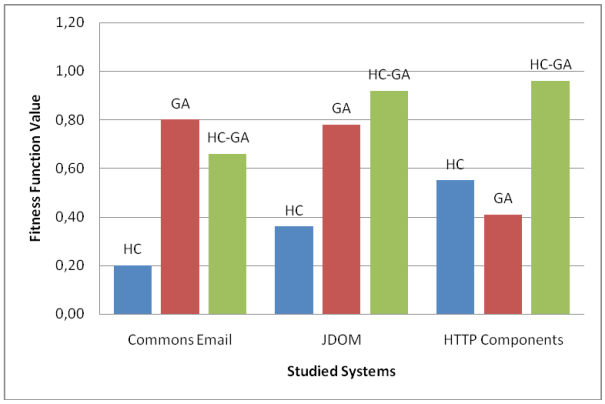

Figure 7: Value of the global fitness function obtained for each algorithm

The combined approach gives a better result than genetic algorithm on a medium system because it starts with an averagely good initial population. This confirms the hypothesis that we have made in section 6.1.

On a large system (HTTP Components), the hierarchical clustering gives better results than genetic algorithm because of the following reasons: Firstly, when the number of classes is large, the probability to randomly generate a good initial population decreases. Thus, the genetic algorithm starts from a bad initial population and it will likely spend more time to reach good solutions. Secondly, the large number of classes and the number of relationships and dependencies between them allow the hierarchical clustering to optimize more efficiently the local fitness function and discover good quality components.

As on medium systems, the combined approach gives better results than genetic algorithm and hierarchical clustering on large systems because it starts from a nearly good initial population.

7. RELATED WORK

Software component identification can be performed in two different manners [4]: top-down and bottom-up. Top-down software component identification is performed by analyzing domain business models to get a set of business components. Bottom-up software component identification is performed by extracting reusable software components from existing software system source code.

Most of the software component identification techniques belong to the first category [12] [14] [16]. This means that they start from semi-formal domain business models (Typically expressed in UML) and produce domain software components. This constitutes an important shortcoming like the inability to apply these approaches when domain business models are missing.

Our approach is based on the analysis of source code which remains the only software artifact that reflects the reality of the system.

Approaches presented in [2] [16] make use of only coupling and cohesion metrics to identify components. However, the

use of only these two metrics can lead to poor quality components. In our approach, we have studied the quality and semantic-correctness of components to guarantee that the identified components will be of high quality. In addition, unlike the previously cited works, our approach has the advantage to allow the architect to give more importance to some sub-characteristics.

In our work we compare two algorithms in the context of component identification. In [1], the authors propose to use genetic algorithm to generate initial population for k-means. Based on [1], the work presented in [7] presents a general purpose empirical comparison between k-means and genetic algorithms.

The work presented in [2] is the closest to ours. The authors propose to combine genetic algorithms and simulated annealing to respectively optimize global and local fitness functions for software identification. It uses dynamic analysis to identify software components from the execution trace of a use case. Its main drawback is the a priori knowledge of the high-level system functionalities. Also, it requires an extensive execution of many execution scenarios to involve all the classes that constitutes the system.

8. CONCLUSION

In this paper, we rely on our previous works [5] [6] to propose, compare and combine two algorithms for software component identification from object- oriented source code. The former is based on hierarchical clustering and the latter is based on genetic algorithms.

We begin by defining a mapping model between objects and components and a measurement model for evaluating semantic-correctness of software component. Based on these models, we formulate the software component identification problem as a partitioning problem.

Then we propose two algorithms for identifying software components from existing systems. The former is based on hierarchical clustering and the latter is based genetic algorithms. We show the limitations of using them separately and we propose to overcome these limitations by combining them to benefit from the advantages of each one. The obtained results by applying the combination of the two algorithms confirm our hypothesis.

As short-term perspective, we plan to extend our approach to apply it on multiple versions/variants of the same system to obtain highly reusable components. The obtained results will guide our long-term perspective which consists in recovering software product lines.

9. REFERENCES

[1] B. Al-Shboul and S.-H. Myaeng. Initializing k-means using genetic algorithms. *World Academy of Science, Engineering and Technology*, 54, 2009.

[2] S. Allier, H. A. Sahraoui, S. Sadou, and S. Vaucher. Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces. In *Proceedings of the 13th international conference on Component-Based Software Engineering*, CBSE'10, pages 216–231, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] J. M. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of the 1995 Symposium on Software reusability*, SSR '95, pages 259–262, New York, NY, USA, 1995. ACM.

[4] D. Birkmeier and S. Overhage. On component identification approaches — classification, state of the art, and comparison. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, CBSE '09, pages 1–18, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] S. Chardigny and A. Seriai. Software architecture recovery process based on object-oriented source code and documentation. In *Proceedings of the 4th European conference on Software architecture*, ECSA'10, pages 409–416, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit. Extraction of component-based architecture from object-oriented systems. In *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, WICSA '08, pages 285–288, Washington, DC, USA, 2008. IEEE Computer Society.

[7] Chittu.V and N.Sumathi. A modified genetic algorithm initializing k-means clustering. *Global Journal of Computer Science and Technology*, 11, 2011.

[8] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[9] G. T. Heineman and W. T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[10] J. H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[11] ISO. Software engineering – Product quality – Part 1: Quality model. Technical Report ISO/IEC 9126-1, International Organization for Standardization, 2001.

[12] H. Jain, N. Chalimeda, N. Ivaturi, and B. Reddy. Business component identification - a formal approach. In *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, EDOC '01, pages 183–, Washington, DC, USA, 2001. IEEE Computer Society.

[13] S. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32:241–254, 1967. 10.1007/BF02289588.

[14] S. D. Kim and S. H. Chang. A systematic method to identify software components. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 538–545, Washington, DC, USA, 2004. IEEE Computer Society.

[15] C. Luer and A. V. D. Hoek. Composition environments for deployable software components. Technical report, 2002.

[16] S. K. Mishra, D. S. Kushwaha, and A. K. Misra. Creating reusable software component from object-oriented legacy system through reverse engineering. *Journal of Object Technology*, 8(5):133–152, 2009.

[17] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.